Nelms Institute for the Connected World
UNIVERSITY of FLORIDA

# Improved Scheduling

## YIELDING, BLOCKING, SLEEPING, IMPROVED SEMAPHORES, DEADLOCKS

# Scheduler Recap

The act of deciding which runnable task is to be executed is called *scheduling*. Formally,

*Given a set of tasks* $T = \{\tau_1, \tau_2, \dots, \tau_n\}$*, a set of processors* $\pi = \{\pi_1, \pi_2, \dots, \pi_m\}$*, and a set of resources* $R = \{R_1, R_2, \dots, R_k\}$*,* scheduling *refers to the act of assigning tasks from* $T$ *to processors from* $\pi$ *and resources from* $R$ *so that all tasks complete under certain imposed constraints.*

# Round Robin Scheduling

Given a set of tasks $T = \{\tau_1, \tau_2, \ldots \tau_n\}$, each task $\tau_i$ is given equal CPU time without regards for priority.

- Start at $\tau_1$ and allow it to run for $P$, then
- Switch to $\tau_2$ and allow it to run for $P$, then
- …
- Switch to $\tau_n$ and allow it to run for $P$, then
- Switch to $\tau_1$ and allow it to run for $P$, then…

# Rate Monotonic Scheduling

- Real time priority scheduler
- The task with the shortest period is scheduled first
- Task is run until it finishes
- Running task is preempted by one with higher priority

# Deadline Monotonic Scheduling

- Also known as *Earliest Deadline First* scheduling
- Real-time priority scheduler
- Attempt to overcome shortcomings of rate monotonic
  - Give priority to tasks that have earliest deadline
  - Higher priority tasks always preempt lower priority tasks

# Producer—Consumer problem

Consider two tasks that:

- Run simultaneously
- Share a common, fixed-size buffer as a queue
- Task $\tau_1$ generates (produces) data and places it into the queue
- Task $\tau_2$ retrieves (consumes) the data and does something with it

→ Processes must be synchronized

  - Using semaphores, for example

# Semaphore Solution

- Track the state of the queue using two semaphores
  - `empty_count`: number of empty places in queue
  - `full_count`: number of elements in queue
- Use binary semaphore (mutex) to ensure queue integrity
  - `use_queue`

# Semaphore Solution

**producer**:

```
P(empty_count)
P(use_queue)
add_to_queue(item)
V(use_queue)
V(full_count)
```

**consumer**:

```
P(full_count)
P(use_queue)
item ← get_from_queue()
V(use_queue)
V(empty_count)
```

- `empty_count` is initialized to the number of slots in the queue
- `full_count` is initially 0
- `use_queue` is initially 1

# Semaphore Implementation

```
/* spinlock wait loop*/
      ldr r2, =lock_address
      mov r1, #1
1:    ldrex r0, [r2]            /* get value of lock, place tag on it */
      cmp r0, #0                /* check if zero */
      bne 1b                    /* if not zero, someone else has lock */
      strex r0, r1, [r2]        /* try to store lock on it, if we lost
                                 * the tag because someone else read
                                 * from it, the store will fail
                                 */
      cmp r0, #0                /* check if the store succeeded */
      bne 1b                    /* if not, we try again */
      /* we now have the lock, access critical resource */
```

# The Issue

```
/* spinlock wait loop*/
        ldr r2, =lock_address
        mov r1, #1
1:      ldrex r0, [r2]
        cmp r0, #0
        bne 1b
        strex r0, r1, [r2]
        cmp r0, #0
        bne 1b
        /* got the lock */
```

- Task, $\tau_i$, is in a loop waiting for the lock to be acquired
- Another task, $\tau_k$, is holding the lock
- $\tau_k$ can not release the lock
- $\tau_i$ will not make any progress during its quantum!
  - $\tau_i$ has a chance to do any useful work in at least two quantums
- CPU time is being wasted!

# The Solution



- Yield the CPU to another task
  - Better CPU utilization
- OS controls scheduling
- Task waiting on lock must notify OS that it wishes to relinquish CPU
  - `sched_yield()`

# Better Semaphore Implementation

```
/* better wait loop*/
      ldr r2, =lock_address
      mov r1, #1
1:    ldrex r0, [r2]          /* get value of lock, place tag on it */
      cmp r0, #0              /* check if zero */
      blne sched_yield        /* we don't have the lock, yield CPU */
      bne 1b                  /* retry to get lock */
      strex r0, r1, [r2]      /* try to store lock on it, if we lost
                               * the tag because someone else read
                               * from it, the store will fail
                               */
      cmp r0, #0              /* check if the store succeeded */
      blne sched_yield        /* we don't have the lock, yield CPU */
      bne 1b                  /* and try again */
      /* we now have the lock, access critical resource */
```

# Yielding the CPU

- Call `sched_yield()` from the task
- Kernel receives message
- Kernel saves current task's context
- Kernel schedules a new task and switches in its context
- Kernel allows new task to execute

# Going Into Kernel

- Need a service from the kernel
  - `svc`: service call instruction
  - Assume that service 5 is the yield service

```
/* Library function wrapper */
extern void __sched_yield(void);
void sched_yield(void) {
   __sched_yield()
}
/* assembly implementation */
   .globl __sched_yield
   .type __sched_yield, %function
__sched_yield:
   svc #5
   bx lr
```

# Round Robin Scheduling

| Task | Period | Arrival | Burst Length |
|---|---|---|---|
| $\tau_1$ | 80 | 0 | 30 |
| $\tau_2$ | 80 | 0 | 15 |

**One CPU Quantum:**

10 ticks

$\tau_2$ will yield after 5 ticks of running until $\tau_1$ ends, lock check is 1 tick

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 | 22 | 24 | 26 | 28 | 30 | 32 | 34 | 36 | 38 |

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 40 | 42 | 44 | 46 | 48 | 50 | 52 | 54 | 56 | 58 | 60 | 62 | 64 | 66 | 68 | 70 | 72 | 74 | 76 | 78 |

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 80 | 82 | 84 | 86 | 88 | 90 | 92 | 94 | 96 | 98 | 100 | 102 | 104 | 106 | 108 | 110 | 112 | 114 | 116 | 118 |

# Round Robin Scheduling

| Task | Period | Arrival | Burst Length |
|---|---|---|---|
| $\tau_1$ | 80 | 0 | 30 |
| $\tau_2$ | 80 | 0 | 15 |

**One CPU Quantum:**

10 ticks

$\tau_2$ will yield after 5 ticks of running until $\tau_1$ ends, lock check is 1 tick

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 4 $\tau_1$ | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 | 22 | 24 | 26 | 28 | 30 | 32 | 34 | 36 | 38 |
| 40 | 42 | 44 | 46 | 48 | 50 | 52 | 54 | 56 | 58 | 60 | 62 | 64 | 66 | 68 | 70 | 72 | 74 | 76 | 78 |
| 80 | 82 | 84 | 86 | 88 | 90 | 92 | 94 | 96 | 98 | 100 | 102 | 104 | 106 | 108 | 110 | 112 | 114 | 116 | 118 |

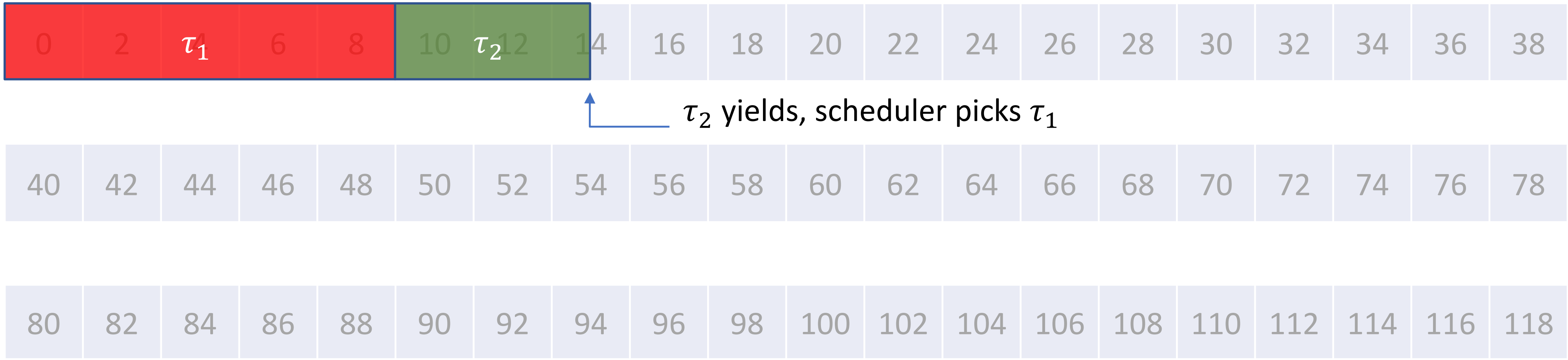
# Round Robin Scheduling

| Task | Period | Arrival | Burst Length |
|---|---|---|---|
| $\tau_1$ | 80 | 0 | 30 |
| $\tau_2$ | 80 | 0 | 15 |

**One CPU Quantum:**

10 ticks

$\tau_2$ will yield after 5 ticks of running until $\tau_1$ ends, lock check is 1 tick



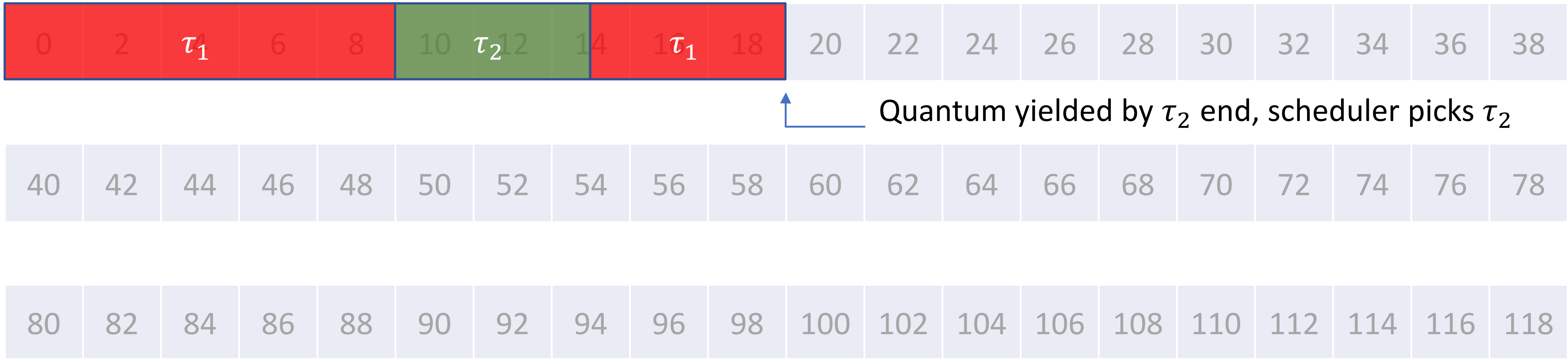
# Round Robin Scheduling

| Task | Period | Arrival | Burst Length |
|---|---|---|---|
| $\tau_1$ | 80 | 0 | 30 |
| $\tau_2$ | 80 | 0 | 15 |

**One CPU Quantum:**

10 ticks

$\tau_2$ will yield after 5 ticks of running until $\tau_1$ ends, lock check is 1 tick



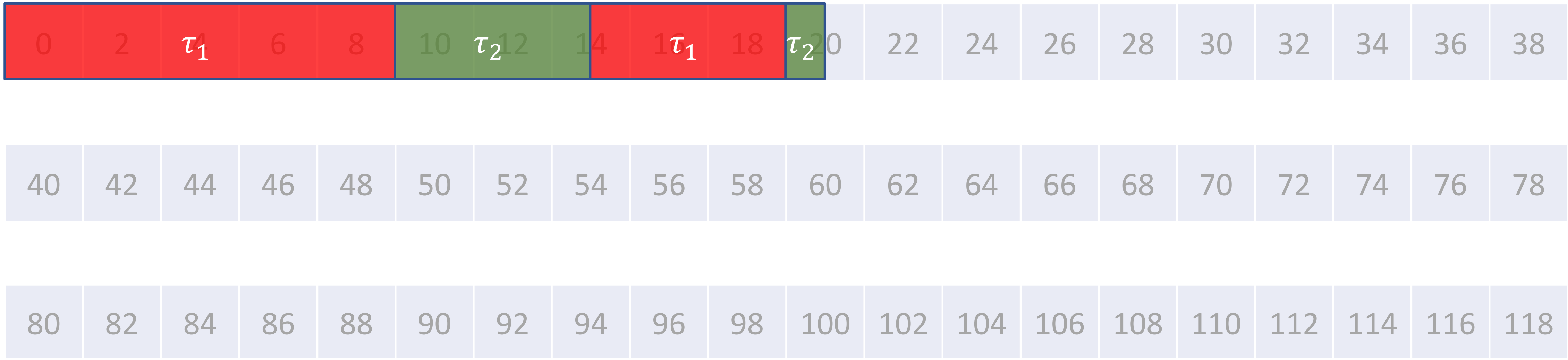
# Round Robin Scheduling

| Task | Period | Arrival | Burst Length |
|---|---|---|---|
| $\tau_1$ | 80 | 0 | 30 |
| $\tau_2$ | 80 | 0 | 15 |

**One CPU Quantum:**

10 ticks

$\tau_2$ will yield after 5 ticks of running until $\tau_1$ ends, lock check is 1 tick



# Round Robin Scheduling

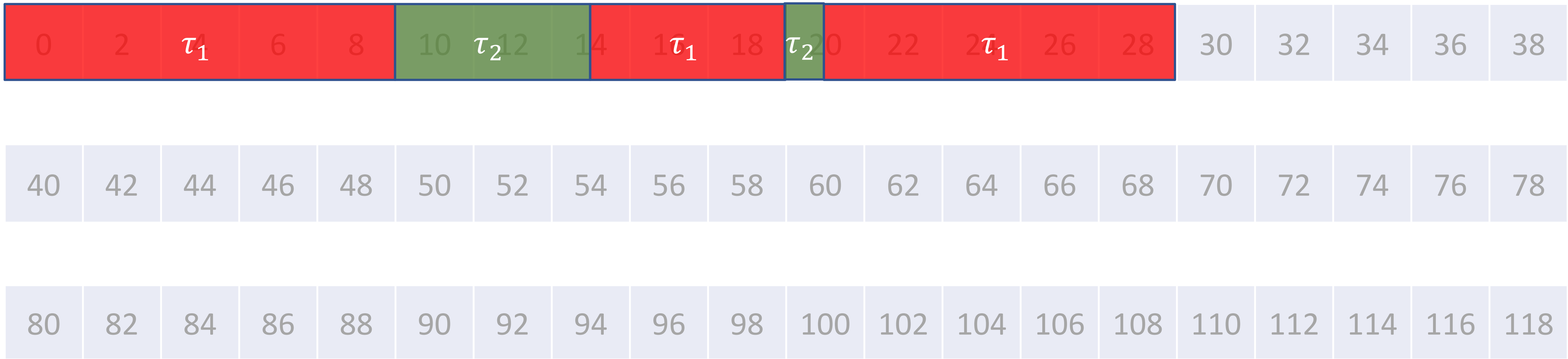


| Task | Period | Arrival | Burst Length |
|---|---|---|---|
| $\tau_1$ | 80 | 0 | 30 |
| $\tau_2$ | 80 | 0 | 15 |

**One CPU Quantum:**

10 ticks

$\tau_2$ will yield after 5 ticks of running until $\tau_1$ ends, lock check is 1 tick



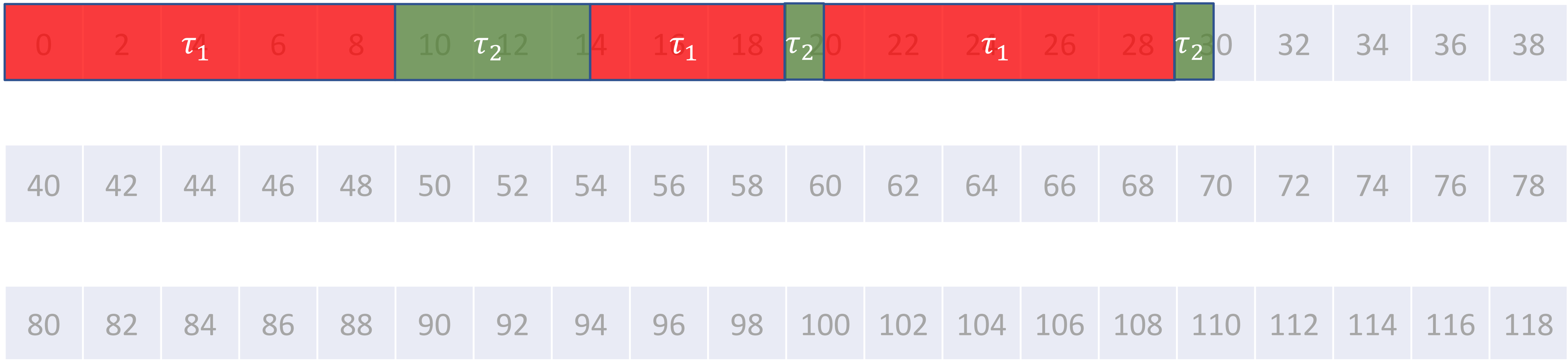
# Round Robin Scheduling

| Task | Period | Arrival | Burst Length |
|---|---|---|---|
| $\tau_1$ | 80 | 0 | 30 |
| $\tau_2$ | 80 | 0 | 15 |

**One CPU Quantum:**

10 ticks

$\tau_2$ will yield after 5 ticks of running until $\tau_1$ ends, lock check is 1 tick



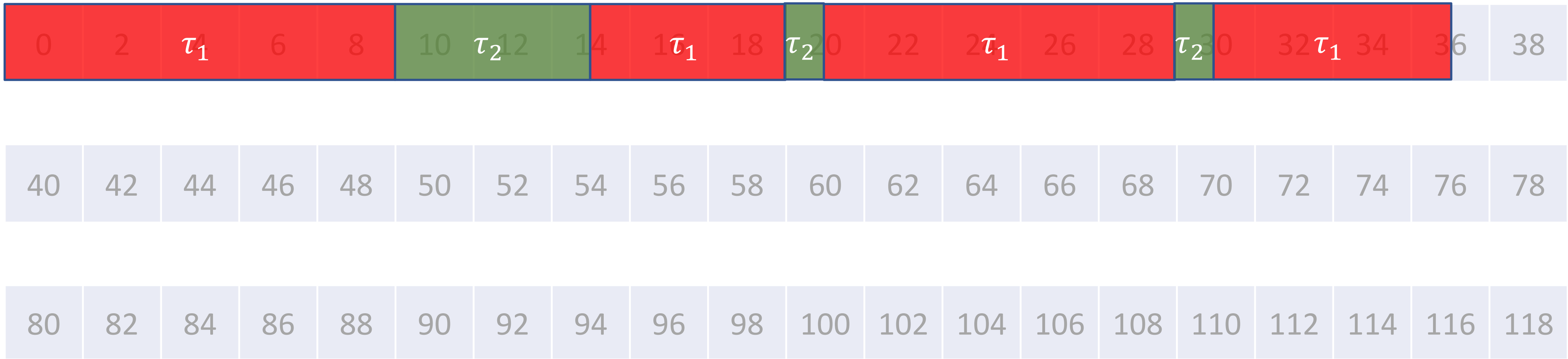
# Round Robin Scheduling

| Task | Period | Arrival | Burst Length |
|---|---|---|---|
| $\tau_1$ | 80 | 0 | 30 |
| $\tau_2$ | 80 | 0 | 15 |

**One CPU Quantum:**

10 ticks

$\tau_2$ will yield after 5 ticks of running until $\tau_1$ ends, lock check is 1 tick



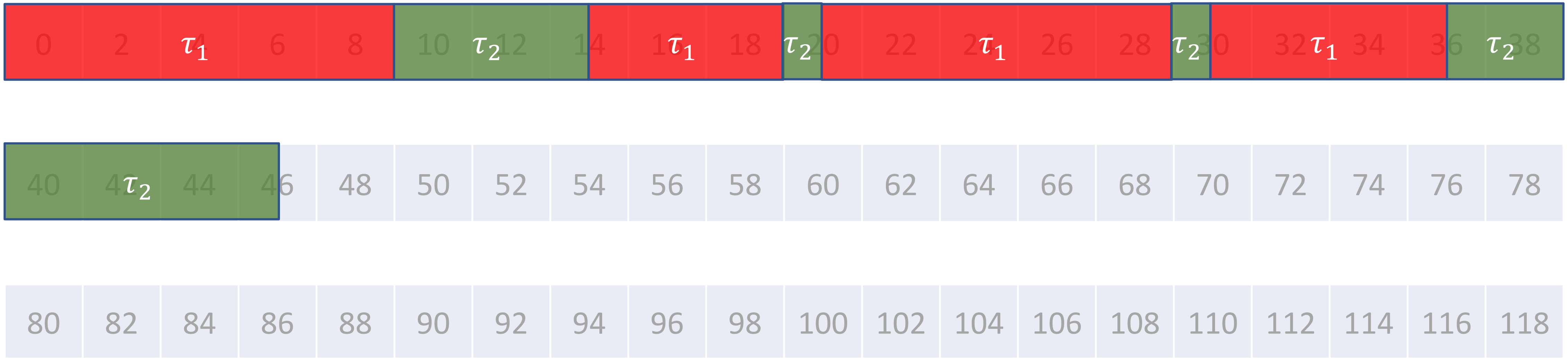
# Round Robin Scheduling

| Task | Period | Arrival | Burst Length |
|---|---|---|---|
| $\tau_1$ | 80 | 0 | 30 |
| $\tau_2$ | 80 | 0 | 15 |

**One CPU Quantum:**

10 ticks

$\tau_2$ will yield after 5 ticks of running until $\tau_1$ ends, lock check is 1 tick



# The Issue

- Task checking and waiting for lock is scheduled
- Yields CPU as soon as it realizes it can't continue
- Task wastes CPU on check
- Scheduler wastes CPU in double context switch
  - Task switches in just to yield
  - Task switches out
  - New task switches in

# Solution

- Block task until lock/resource is free
- Task is flagged as TASK_BLOCKED
- Scheduler will not switch into TASK_BLOCKED tasks
- OS needs to know lock/resource task is waiting for

  → Need to tell kernel what resource we are waiting on

  → Lock holder needs to tell kernel that resource has been released

# Resources and Locks

- Locks do not necessarily have to refer to mutexes/semaphores
- Locks can also be applied to hardware resources
  - Example: the I2C interface in your boards that connect to the LED driver
  - Only one task should be able to access it at a time
  - If a task doing an I2C write is preempted in the middle of sending a command stream, and the new task attempts to send its own, the LED driver will be misconfigured for both tasks.

# Notifying the Kernel

- Need to tell the kernel
  - Resource we are using
  - Action on resource [grab/release/wait]
- Use service call interface
  - `sys_futex`: fast userspace mutexes
  - `sys_futex(void* resource, int action);`

# About `sys_futex`

```
sys_futex:
    if action is wait:
        add resource to task
        block task
    if action is grab:
        if resource is held:
            add resource to task
            block task
        else
            set resource as used
    if action is release:
        unblock resource
        unblock one blocked task waiting on resource
```

# Blocking Semaphore Implementation

```
/* blocking wait loop*/
        ldr r0, =lock_address
        ldr r1, =WAIT
        mov r2, #1
1:      ldrex r3, [r0]              /* get value of lock, place tag on it */
        cmp r3, #0                  /* check if zero */
        blne futex                  /* we don't have the lock, block */
        bne 1b                      /* retry to get lock */
        strex r3, r2, [r0]          /* try to store lock on it, if we lost
                                     * the tag because someone else read
                                     * from it, the store will fail
                                     */
        cmp r3, #0                  /* check if the store succeeded */
        blne futex                  /* we don't have the lock, block */
        bne 1b                      /* and try again */
        /* we now have the lock, access critical resource */
        ldr r1, =GRAB               /* tell system we have the lock */
        bl futex
```

# Round Robin Scheduling

| Task | Period | Arrival | Burst Length |
|---|---|---|---|
| $\tau_1$ | 80 | 0 | 30 |
| $\tau_2$ | 80 | 0 | 15 |

**One CPU Quantum:**

10 ticks

$\tau_2$ will yield after 5 ticks of running until $\tau_1$ ends, lock check is 1 tick

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 | 22 | 24 | 26 | 28 | 30 | 32 | 34 | 36 | 38 |

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 40 | 42 | 44 | 46 | 48 | 50 | 52 | 54 | 56 | 58 | 60 | 62 | 64 | 66 | 68 | 70 | 72 | 74 | 76 | 78 |

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 80 | 82 | 84 | 86 | 88 | 90 | 92 | 94 | 96 | 98 | 100 | 102 | 104 | 106 | 108 | 110 | 112 | 114 | 116 | 118 |

# Round Robin Scheduling

| Task | Period | Arrival | Burst Length |
|---|---|---|---|
| $\tau_1$ | 80 | 0 | 30 |
| $\tau_2$ | 80 | 0 | 15 |

**One CPU Quantum:**

10 ticks

$\tau_2$ will yield after 5 ticks of running until $\tau_1$ ends, lock check is 1 tick

$\tau_1$

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 | 22 | 24 | 26 | 28 | 30 | 32 | 34 | 36 | 38 |

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 40 | 42 | 44 | 46 | 48 | 50 | 52 | 54 | 56 | 58 | 60 | 62 | 64 | 66 | 68 | 70 | 72 | 74 | 76 | 78 |

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 80 | 82 | 84 | 86 | 88 | 90 | 92 | 94 | 96 | 98 | 100 | 102 | 104 | 106 | 108 | 110 | 112 | 114 | 116 | 118 |

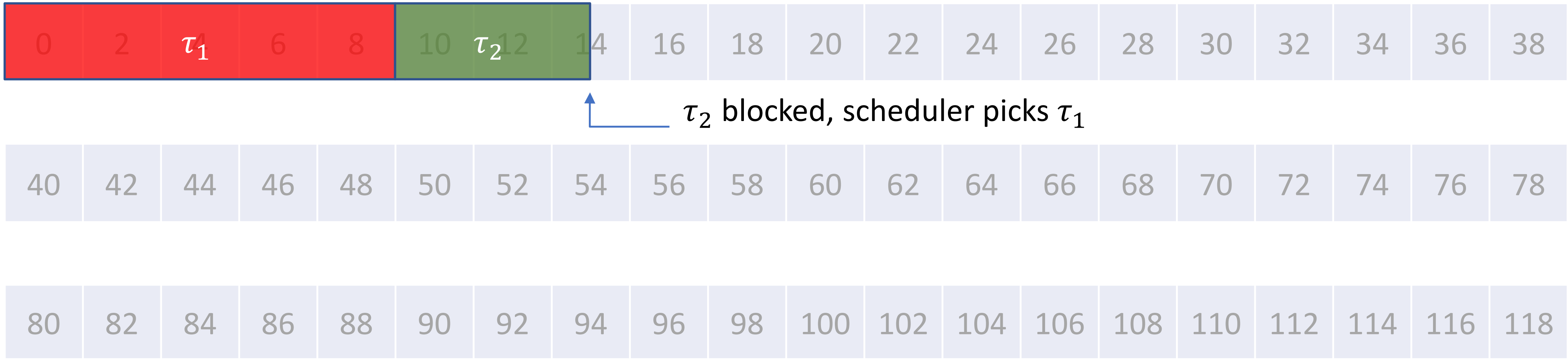
# Round Robin Scheduling

| Task | Period | Arrival | Burst Length |
|---|---|---|---|
| $\tau_1$ | 80 | 0 | 30 |
| $\tau_2$ | 80 | 0 | 15 |

**One CPU Quantum:**

10 ticks

$\tau_2$ will yield after 5 ticks of running until $\tau_1$ ends, lock check is 1 tick



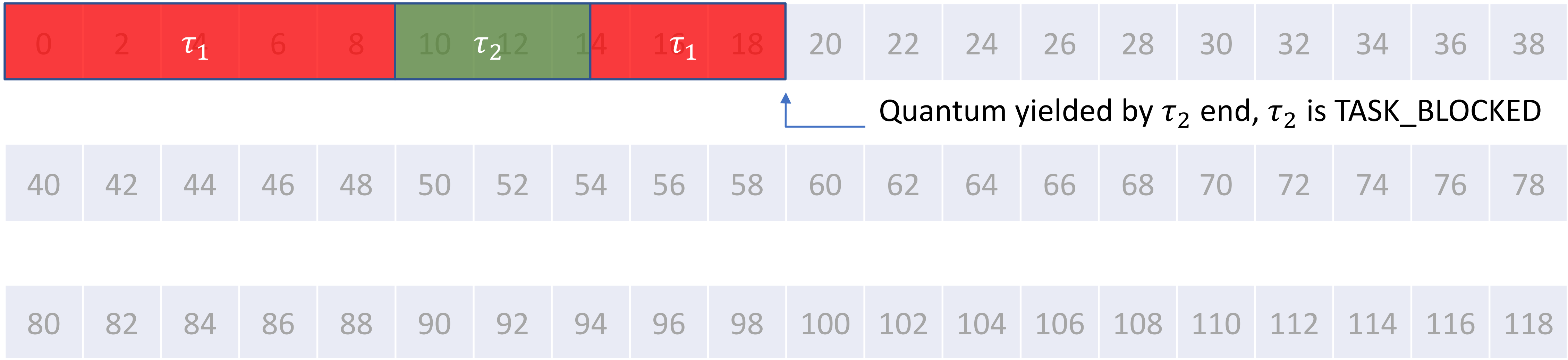
# Round Robin Scheduling

| Task | Period | Arrival | Burst Length |
|---|---|---|---|
| $\tau_1$ | 80 | 0 | 30 |
| $\tau_2$ | 80 | 0 | 15 |

**One CPU Quantum:**

10 ticks

$\tau_2$ will yield after 5 ticks of running until $\tau_1$ ends, lock check is 1 tick



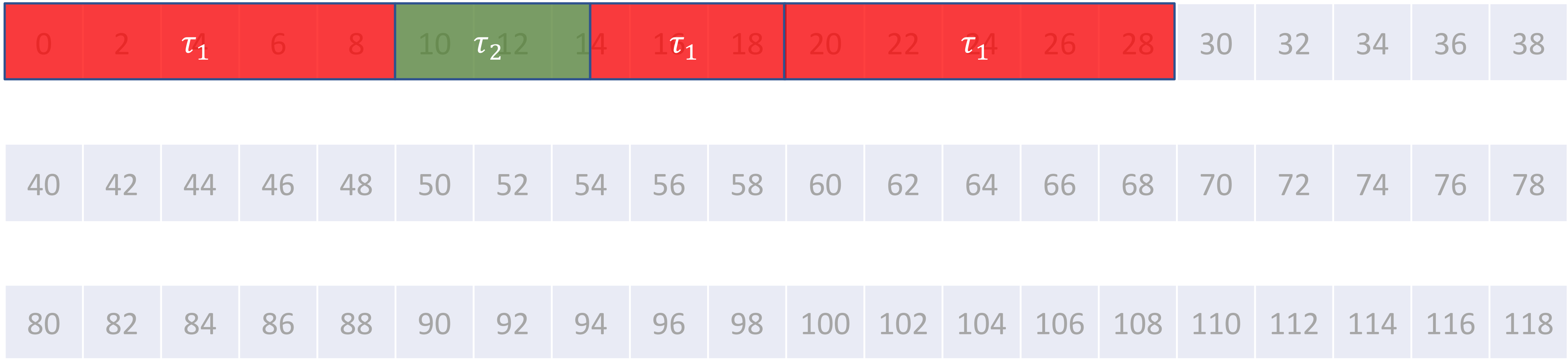
# Round Robin Scheduling

| Task | Period | Arrival | Burst Length |
|---|---|---|---|
| $\tau_1$ | 80 | 0 | 30 |
| $\tau_2$ | 80 | 0 | 15 |

**One CPU Quantum:**

10 ticks

$\tau_2$ will yield after 5 ticks of running until $\tau_1$ ends, lock check is 1 tick



# Round Robin Scheduling

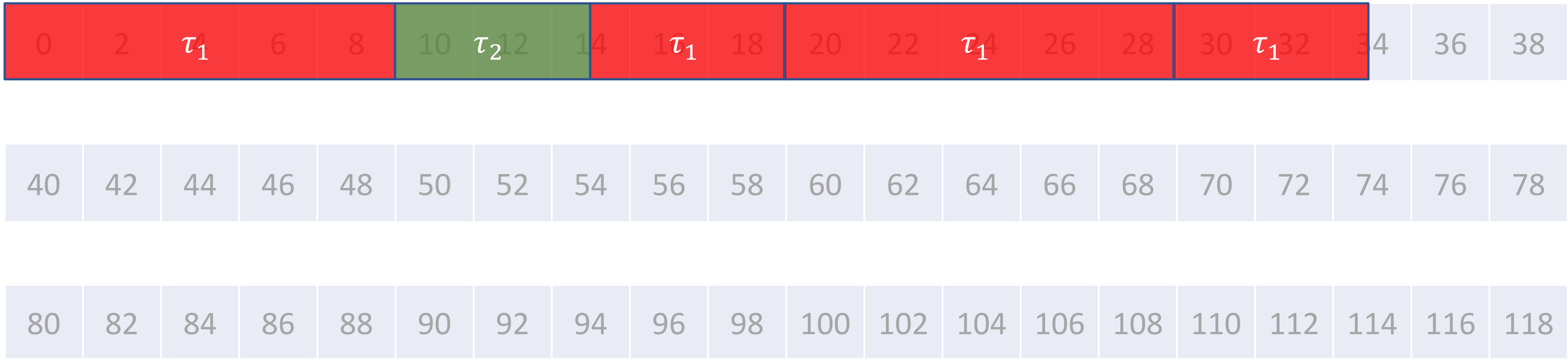


| Task | Period | Arrival | Burst Length |
|---|---|---|---|
| $\tau_1$ | 80 | 0 | 30 |
| $\tau_2$ | 80 | 0 | 15 |

**One CPU Quantum:**

10 ticks

$\tau_2$ will yield after 5 ticks of running until $\tau_1$ ends, lock check is 1 tick



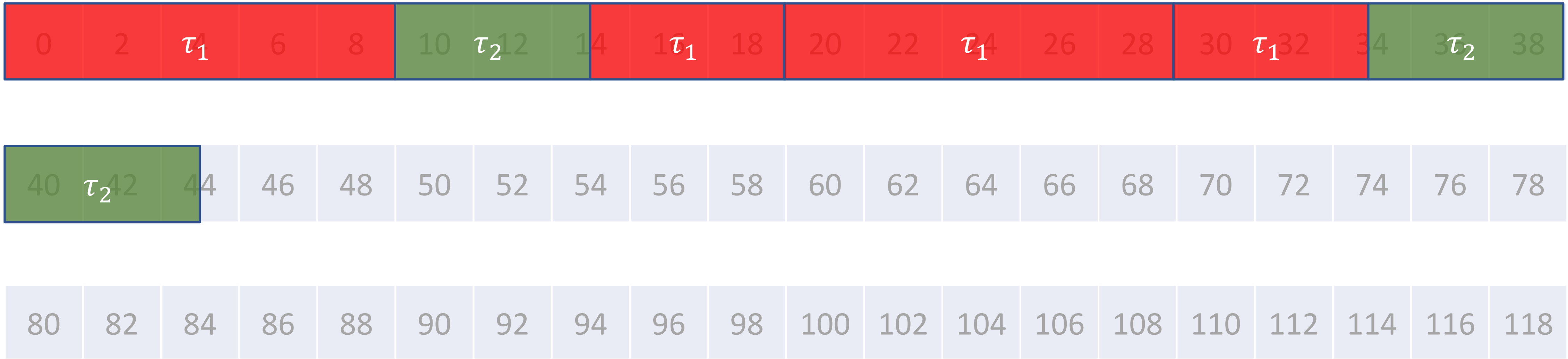
# Round Robin Scheduling

| Task | Period | Arrival | Burst Length |
|---|---|---|---|
| $\tau_1$ | 80 | 0 | 30 |
| $\tau_2$ | 80 | 0 | 15 |

**One CPU Quantum:**

10 ticks

$\tau_2$ will yield after 5 ticks of running until $\tau_1$ ends, lock check is 1 tick



# Multiple Locks

- Tasks may have to request multiple locks
- Order of request is important
- May lead to unresponsive systems

# Eating and Philosophy

Five philosophers sit at a dining table. Each one of them is given a[n infinite] bowl of food. Between each bowl, a fork is placed. A philosopher can either eat or speak, but can not do both at once. In order to eat, a philosopher must be holding both the fork to the right and to the left of their bowl. A philosopher can not eat with only one fork.

Assuming that the philosophers do not know when the other wish to eat or speak, design an algorithm that allows the philosophers to eat and speak forever.

# Dining Philosophers Problem

Attempted solution:

- Speak until left fork is available, take it when it is
- Speak until right fork is available, take it when it is
- Eat for a fixed amount of time
- Place right fork on table
- Place left fork on table
- Repeat from the start

# Dining Philosophers Problem

Issue with solution:

- What happens if all philosophers take the first action at the same time?

# Consider the Scenario

- Two tasks are running, $\tau_1$ and $\tau_2$
- Task $\tau_1$ acquires lock $L_1$
- $\tau_1$ is preempted, $\tau_2$ starts to run
- $\tau_2$ acquires lock $L_2$
- $\tau_2$ is preempted, $\tau_1$ resumes
- $\tau_1$ attempts to acquire lock $L_2$, lock is in use, task is blocked
- $\tau_2$ resumes, tries to acquire lock $L_1$, lock in use, task is blocked

→ Tasks $\tau_1$ and $\tau_2$ are stuck waiting on each other!

→ **System is deadlocked!**

# Deadlocks

- *Coffman Conditions*
  - *Mutual Exclusion*: resources are unshareable
  - *Hold and wait*: task currently holding a resource is requesting a resource held by another task
  - *No preemption*: resources can only be forfeited voluntarily by tasks
  - *Circular wait:* In a set of tasks $T = \{\tau_1, \tau_2, \dots, \tau_n\}$ task $\tau_1$ is waiting on a resource held by $\tau_2$, who waits on a resource held by task $\tau_3$ … until $\tau_n$ which waits on a resource held by task $\tau_1$.

# Detecting Deadlocks

- Track resource allocation
- Track process states
- Attempt to make prediction based on information

→ Deadlock may still go by undetected

# Dealing with Deadlocks

- Preempt resources being held by task
  - Task may behave erratically afterwards
- Terminate task holding resource
  - Breaks circular wait
  - Work done by task lost
  - Functionality lost
  - May lead to system instability

# Preventing Deadlocks

Eliminate one of the four Coffman conditions

- *mutual exclusion*
  - Tasks have spooled access to resources
  - Tasks can’t have locks on resources
  - Some resources are non-trivial to spool
  - Deadlocks can still arise
- *hold and wait*
- *preempt resources*
- *circular wait*

# Preventing Deadlocks

Nelms Institute for the Connected World
UNIVERSITY *of* FLORIDA

Eliminate one of the four Coffman conditions

- *mutual exclusion*
- *hold and wait*
  - Tasks request all needed resources at startup
  - Difficult to predict resource utilization
  - Wastes resources
- *preempt resources*
- *circular wait*

# Preventing Deadlocks

Eliminate one of the four Coffman conditions

- *mutual exclusion*
- *hold and wait*
  - Tasks can only request resources when they are holding none
  - Impractical at times
  - Long wait times for commonly used resources (resource starvation)
- *preempt resources*
- *circular wait*

# Preventing Deadlocks

Eliminate one of the four Coffman conditions

- *mutual exclusion*
- *hold and wait*
- *preempt resources*
    - Allow resource preemption
    - May be impossible
    - Task may require to hold the resource or result may be inconsistent
- *circular wait*

# Preventing Deadlocks

Eliminate one of the four Coffman conditions

- *mutual exclusion*
- *hold and wait*
- *preempt resources*
- *circular wait*
  - Determine partial ordering of resources using hierarchy
  - Graph traversal problem
  - Issue in constructing graph
  - Solution not obvious

# Modern OS & Deadlocks

This is what most Operating Systems do to deal with deadlocks:

# Modern OS & Deadlocks

This is what most Operating Systems do to deal with deadlocks:

Assume they do not happen.

# Modern OS & Deadlocks

This is what most Operating Systems do to deal with deadlocks:

It is the programmer's responsibility to ensure no deadlock conditions occur. If one does occur, kill offending tasks manually.

# Modern OS & Deadlocks

This is what most Operating Systems do to deal with deadlocks:

If the OS is deadlocked, hit the reset button.

# Idling Tasks

- Sometimes we need to delay execution
  - Need to wait on a condition
  - e.g. wait for device to respond to command
- Could use an idle delay loop
  - Decrement a variable until it reaches 0, doing nothing.

# Issue & Solution

- Executing task is idling, doing nothing
- CPU is wasted, other tasks could be doing something
- Issue exacerbated if we are scheduled again in the same idle loop

→ Put the task to sleep!

- Notify kernel we wish to sleep
- Kernel adds timer to task and blocks it
  - On every SysTick event, decrement timer
  - If timer reaches 0, unblock task

# The sys_usleep Service Call

```
void sys_usleep(unsigned int us) {
      struct timer_list t;
      save_current_context();              /* save task context */
      t = get_new_timer();                 /* get a new timer */
      t->count = us;                       /* set its value */
      t->state = ACTIVE;                   /* set it to active */
      t->task = current;                   /* attach current task */
      current->state  = TASK_BLOCKED;      /* block current task */
      add_timer(timers, t);                /* add the timer to list */
      schedule();                          /* schedule a new task */
      restore_context();                   /* restore its context */
}
```

# SysTick and Timers



```
void systick_handler(void) {
        /* ... */
        for(timer_list* t = timers; t; t = t->next) {
                t->count--;
                if(!t->count) {
                        t->task->state = TASK_RUNNABLE;
                        t->state = EXPIRED;
                }
        }
        remove_expired_timers(timers);
        /* handle other SysTick events */
}
```

# Effect

- Kernel will not schedule task while it is sleeping
- Sleep resolution dependent on SysTick interrupt frequency
  - Not guaranteed to be exact
  - Make SysTick too fast and CPU will be wasted on servicing that interrupt
  - 100 us – 10ms resolution ok for most cases

# Crash Course: C

DATA STRUCTURES: CIRCULAR BUFFERS AND FIFOS

# Circular Buffer

- Normal buffers are fixed in size, with one index element
  - Think arrays

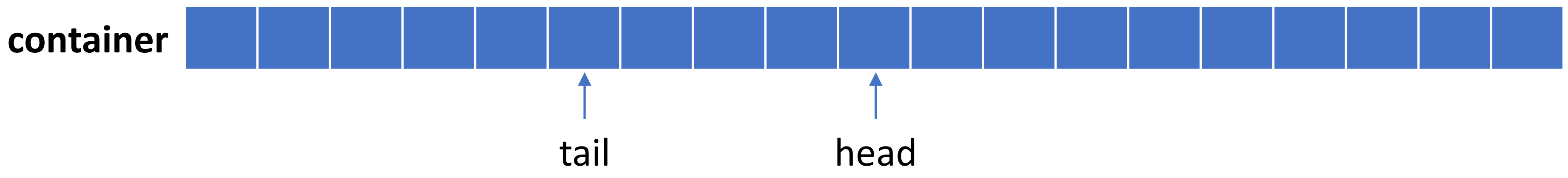

```c
/* C version */
#define N 16
struct buffer_int {
   int container[N];
   size_t index;
};
```

```cpp
/* C++ version */
template<typename T, size_t N>
struct buffer {
   T container[N];
   size_t index;
};
```

# Circular Buffer

UF Nelms Institute for the Connected World
UNIVERSITY of FLORIDA

- Circular buffers are also fixed in size, however
  - They contain two pointer elements: a head and a tail
  - They optionally contain a count of used elements




```c
/* C version */
#define N 16
struct buffer_int {
   int container[N];
   size_t head, tail, count;
};
```

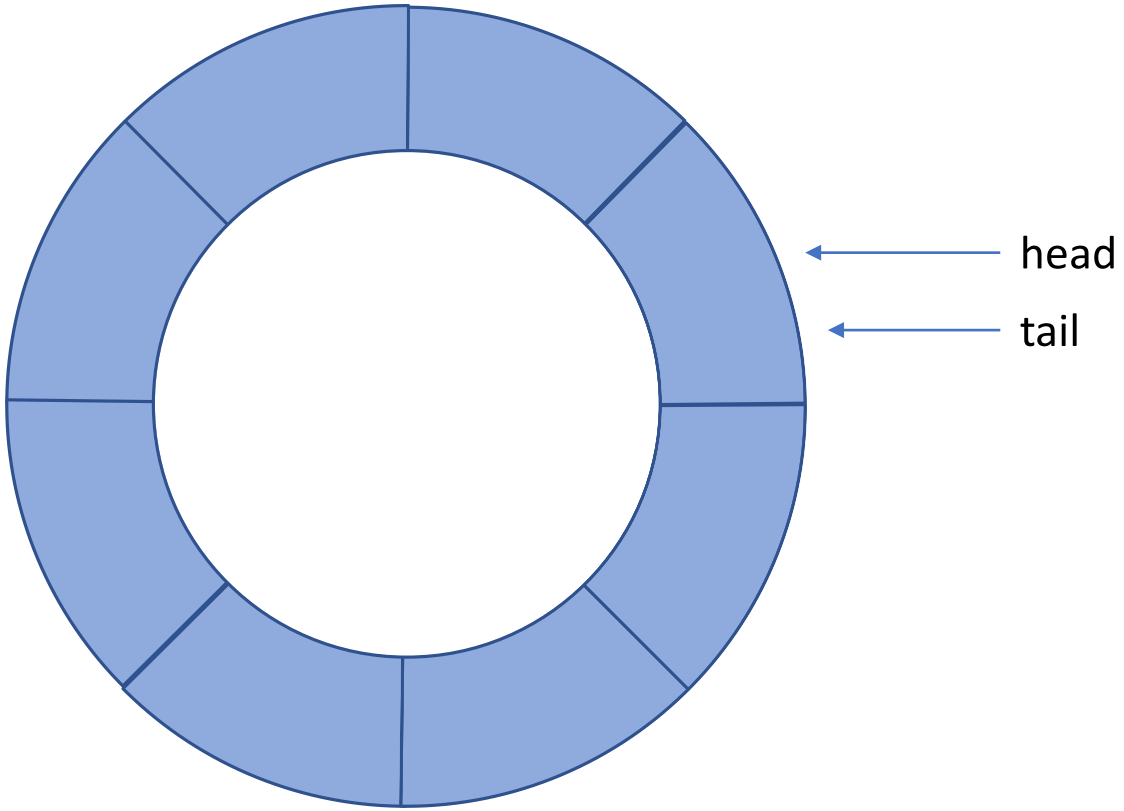
```cpp
/* C++ version */
template<typename T, size_t N>
struct buffer {
   T container[N];
   size_t head, tail, count;
};
```

# Circular Buffer

- Operations:
  - Put element: write at the head of the buffer
  - Get element: read at the tail of the buffer
- Initialization:
  - Set head, tail to start of container
  - Set count to 0
- When head reaches end of container
  - Reset head to initial position
  - Start adding elements at the start of the buffer
  - But only if there’s space on the buffer (count < N)

# Circular Buffer

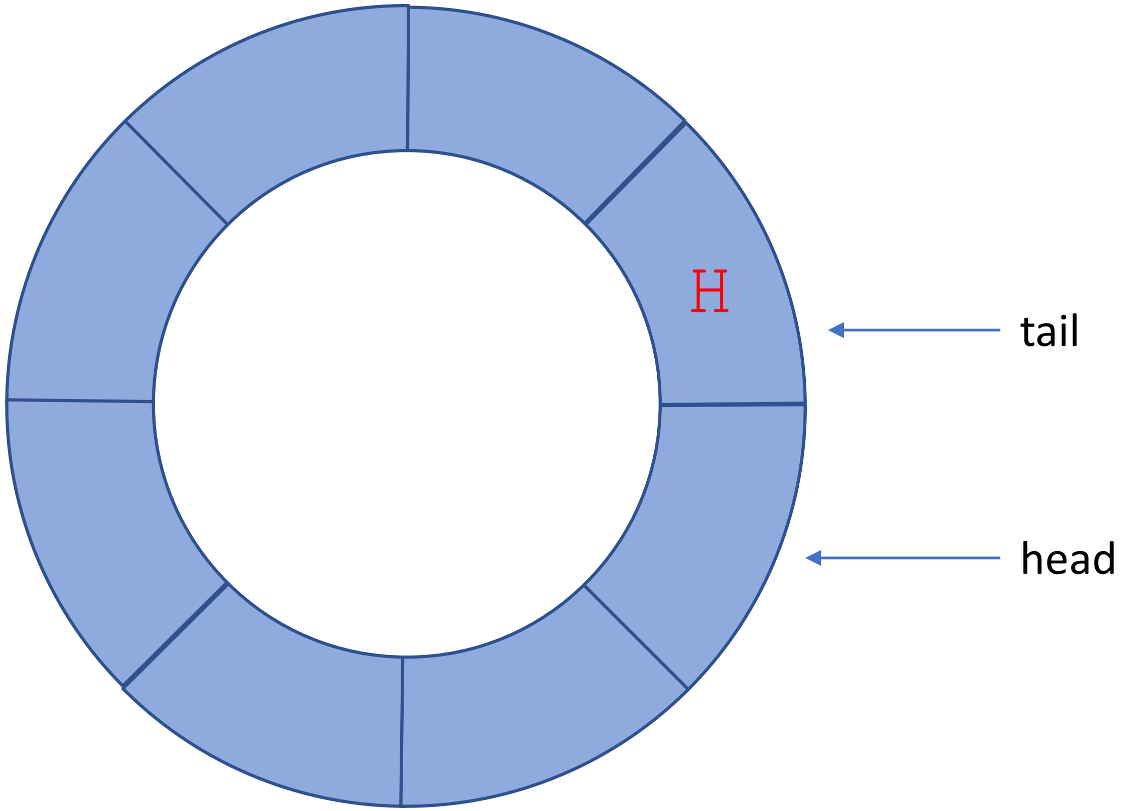
```
Hello, world!
```



# Circular Buffer

`Hello, world!`

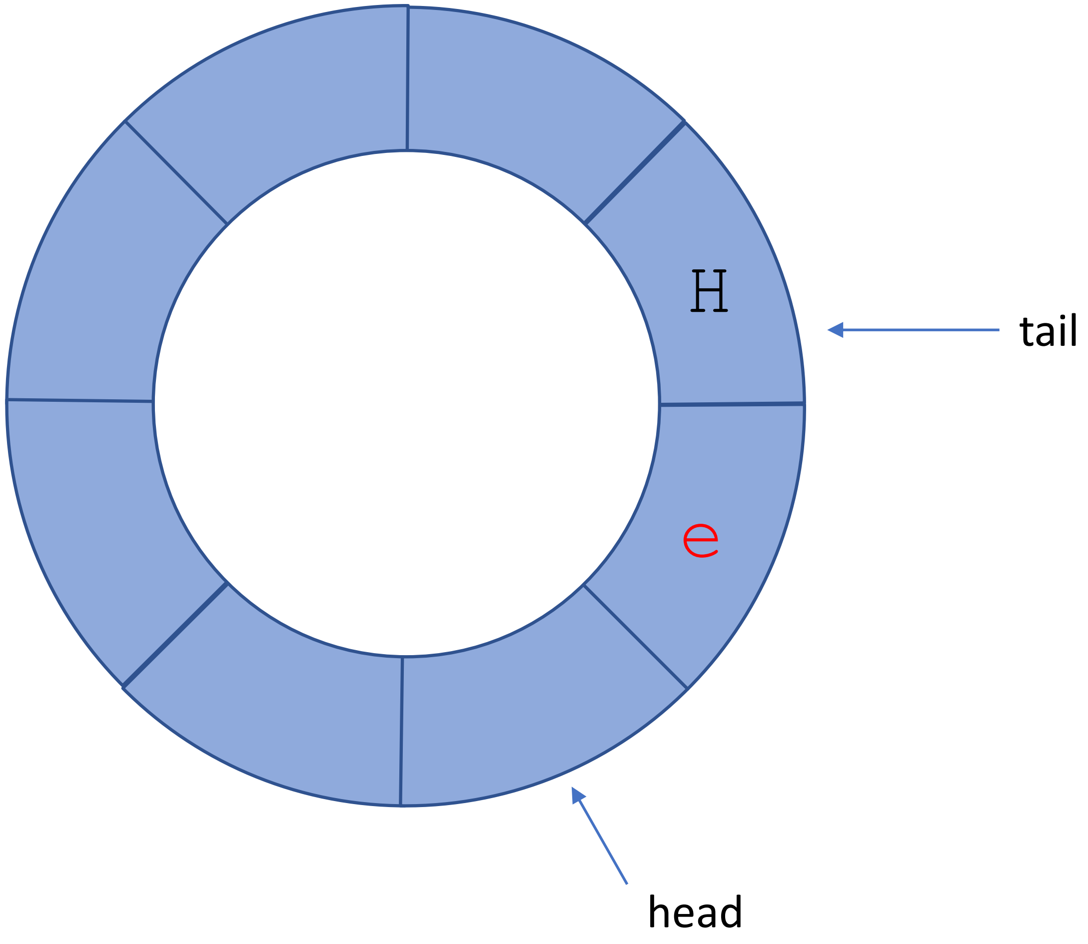
**Operation:** put



# Circular Buffer

`Hello, world!`

**Operation:** put

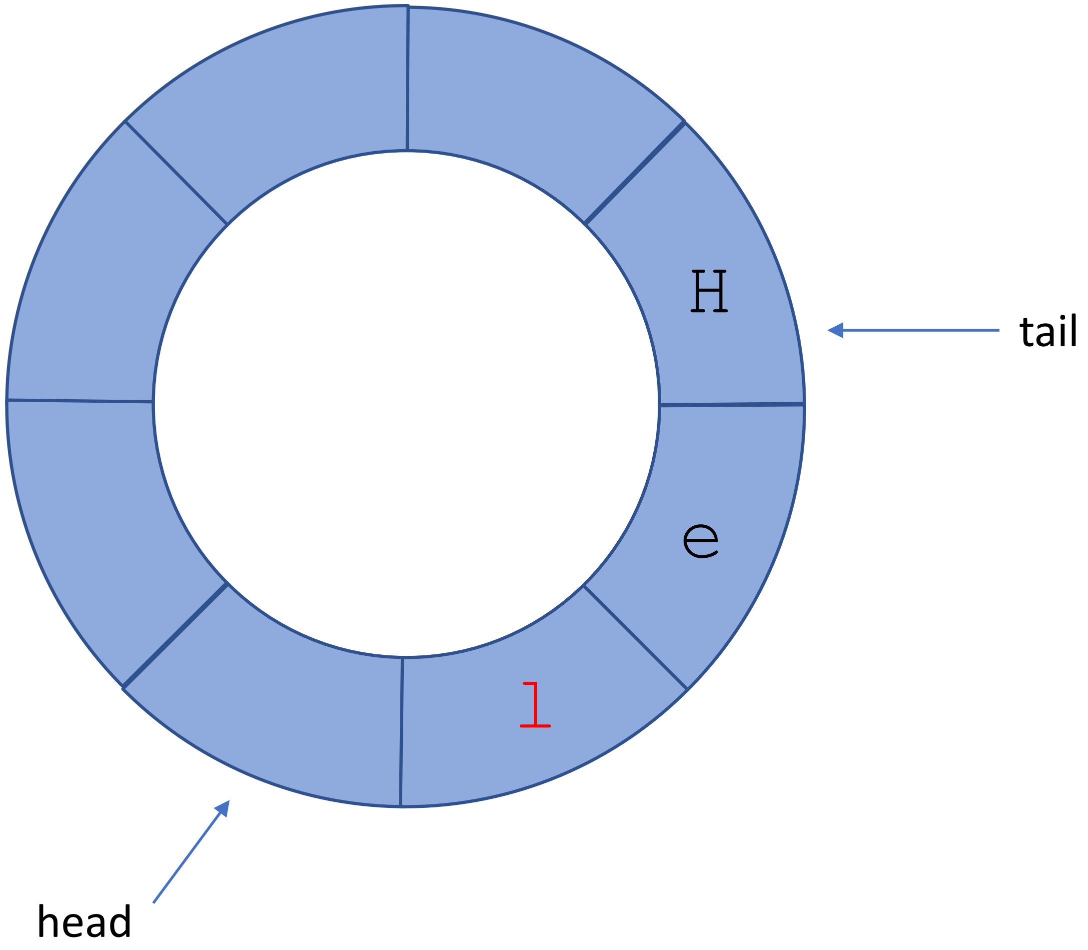


# Circular Buffer



`Hello, world!`

**Operation:** put

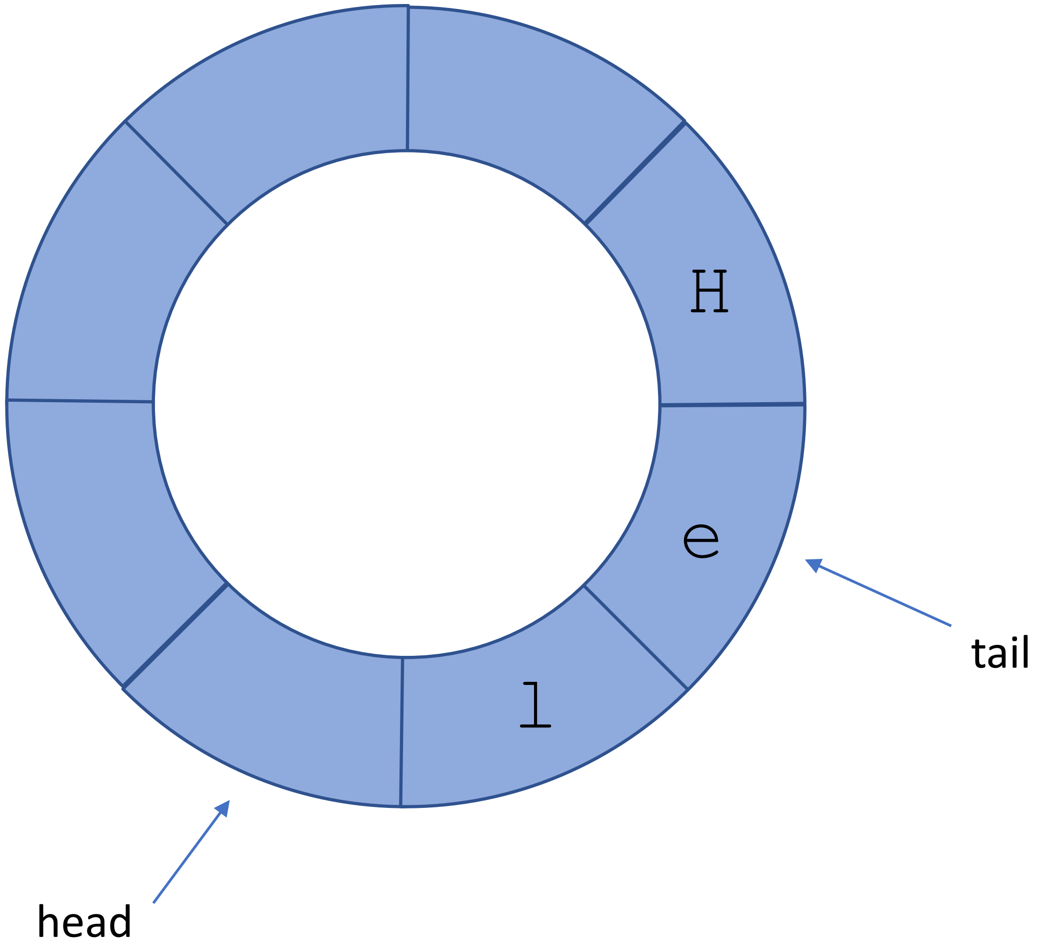


# Circular Buffer

`Hello, world!`

**Operation:** get

H



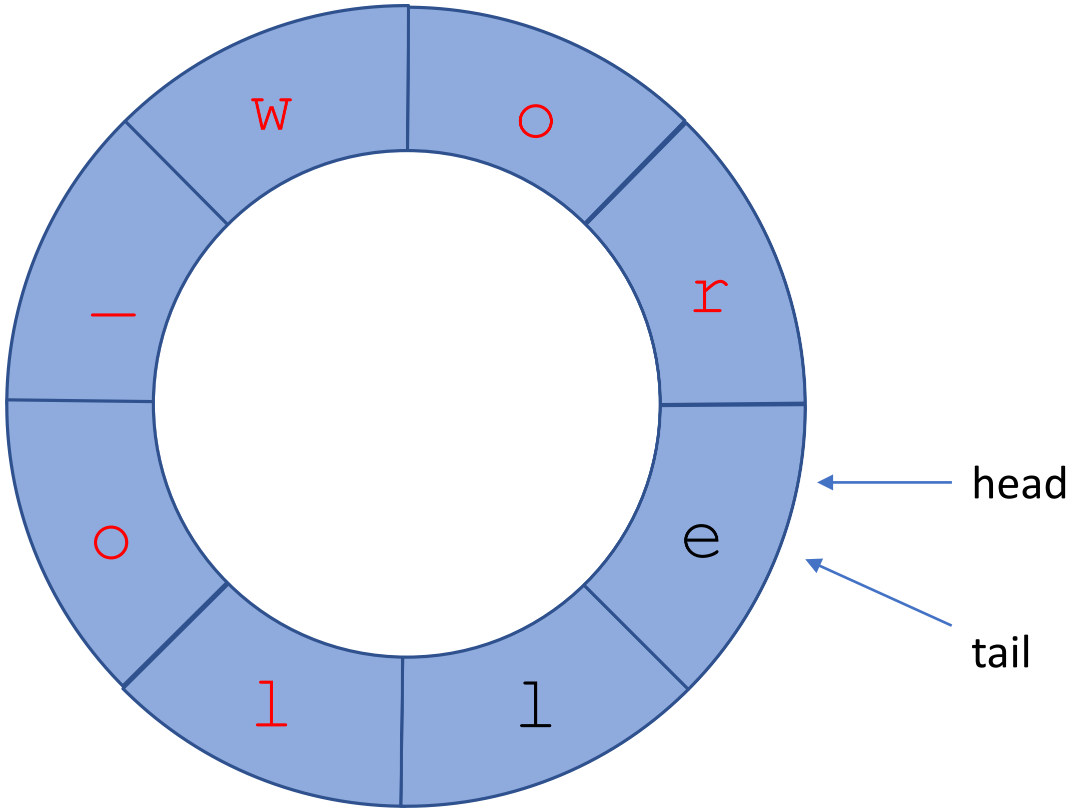
# Circular Buffer



`Hello, world!`

**Operation:** put x 7

`H`



# Circular Buffer

```
/* circular_buffer.h */
struct circular_buffer;

struct circular_buffer* cb_put(struct circular_buffer* cb, char c);
struct circular_buffer* cb_get(struct circular_buffer* cb, char* c);
int cb_is_full(struct circular_buffer* cb);
int cb_is_empty(struct circular_buffer* cb);
```

# Circular Buffer

```c
/* circular_buffer.c */
#define N 16
struct circular_buffer {
      char container[N];
      size_t head, tail, count;
};

int cb_is_full(struct circular_buffer* cb) {
      return cb->count == N;
}

int cb_is_empty(struct circular_buffer* cb) {
      return !cb->count;
}
```

# Circular Buffer

```
/* circular_buffer.c */
#define N 16
struct circular_buffer* cb_put(struct circular_buffer* cb, char c) {
	if(!cb || cb_is_full(cb)) {
		return (struct circular_buffer*)NULL;
	}
	cb->container[cb->head++];
	cb->head &= (N - 1);
	cb->count++;
	return cb;
}
```

# Circular Buffer

```c
/* circular_buffer.c */
struct circular_buffer* cb_get(struct circular_buffer* cb, char* c) {
      if(!cb || cb_is_empty(cb)) {
            return (struct circular_buffer*)NULL;
      }
      *c = cb->container[cb->tail++];
      cb->tail &= (N - 1);
      cb->count--;
      return cb;
}
```

# Circular Buffer

- First element added to the buffer is the first one to be removed
- Circular buffers belong to a category of data structures called FIFOs
  - First-In First-Out data structures
- FIFOs can be implemented using other data structures
  - e.g. Linked lists: put at tail, get at head

# Inter-Process Communication

SHARED MEMORY, SOCKETS, FILES, PIPES

# Motivation

- Tasks normally run isolated from each other
- Necessary to two or more tasks to share information
  - Example: a task gathers GPS coordinate, another task processes the coordinates
- Tasks must synchronize data transmission

# IPC Mechanisms

- Shared Memory
- Sockets
- Files
- Pipes

# IPC Mechanisms

- Shared Memory
  - OS gives access to same region of memory to different tasks
  - One task writes to the shared memory area
  - Another task can read the written information
  - Tasks are responsible for creating synchronization primitives
  - Equivalent to a shared buffer
- Sockets
- Files
- Pipes

# IPC Mechanisms



- Shared Memory
- Sockets
  - Data is sent over a network interface
  - Socket may be local and created over a virtual network interface (e.g. loopback)
  - Tasks must manage the communications protocol
- Files
- Pipes

# IPC Mechanisms

- Shared Memory
- Sockets
- Files
  - Record in a storage device
  - Tasks open the same file
  - Perform read/write operations on file to receive/send data
  - OS *must* ensure file reads and writes are atomic for consistency
- Pipes

# IPC Mechanisms

- Shared Memory
- Sockets
- Files
- Pipes
  - Unidirectional data channel
  - Read/writes are handled by OS
  - Data written to one end of the pipe is buffered by the OS
  - Data read from the other end of the pipe is removed from buffer

# Implementing Pipes

```
/* We start by declaring a file descriptor structure. The file
descriptor contains a pointer to its corresponding write function, its
corresponding read function, a close function, a function to set
attributes, a pointer to the underlying data structure containing file
descriptor information, and the file descriptor number */
struct file_descriptor {
      ssize_t (*read_fn)(void*, void*, size_t);
      ssize_t (*write_fn)(void*, const void*, size_t);
      int (*close_fn)(void*);
      int (*fcntl_fn)(void*, int, ...);
      void* fd_struct_data;
      int fildes;
};
```

# Implementing Pipes

```
/* We also define the pipe as a circular buffer, but add an entry for
flags. This allows us to record information about the pipe. */
#define PIPE_BUF_LEN    32
struct pipe_struct {
      char container[PIPE_BUF_LEN];
      size_t head, tail, count;
      int flags[2];
};
```

# Implementing Pipes

```
/* We assume the existence of the following functions. Their
implementation are not given here. */
/** obtains a pointer to the next available file descriptor struct */
struct file_descriptor* get_next_fildes(void);
/** obtains a pointer to the next available pipe struct */
struct pipe_struct* get_next_pipe(void);
/** releases a file descriptor struct to the kernel */
void release_fildes(struct file_descriptor* fd);
/** obtain a file descriptor struct associated with current process */
struct file_descriptor* get_fildes(int fd);
/** places a task to sleep waiting for an event */
void interruptible_wait(void);
```

# Implementing Pipes

```
/* service call to create a pipe. */
int sys_pipe(int pipefd[2]) {
        struct file_descriptor* fd[2];
        struct pipe_struct* ps;

        if(!(fd[0] = get_next_fildes())) {
                /* no more file descriptors available */
                goto err_no_fildes_0;
        }

        /* continues... */
```

# Implementing Pipes

```
/* continued */
if(!(fd[1] = get_next_fildes())) {
        /* no more file descriptors available */
        goto err_no_fildes_1;
}

if(!(ps = get_next_pipe())) {
        /* no more pipe descriptors available */
        goto err_no_pipe;
}

/* continues... */
```

# Implementing Pipes

```
/* continued */
/* read end of pipe */
fd[0]->read_fn = pipe_read;
fd[0]->fd_struct_data = ps;

/* write end of pipe */
fd[1]->write_fn = pipe_write;
fd[1]->fd_struct_data = ps;

pipefd[0] = file_descriptor[0]->fildes;
pipefd[1] = file_descriptor[1]->fildes;
/* continues... */
```

# Implementing Pipes

```
        /* continued */
        return 0;
        /* error handling */
err_no_pipe:
        release_fildes(fd[1]);
err_no_fildes_1:
        release_fildes(fd[0]);
err_no_fildes_0:
        return -ENFILE;
}
```

# Implementing Pipes

```
/* read from pipe */
ssize_t pipe_read(void* fd_struct, void* buf, size_t count) {
      struct pipe_struct* p = (struct pipe_struct*)fd_struct;
      size_t ret = 0;
      while(ret < count &&
                  (!cb_is_empty(p) || !(p->flags[0] & O_NONBLOCK)) {
            while(cb_is_empty(p)) {
                  /* blocking read, wait for data */
                  interruptible_wait();
            }
            cb_get(p, (((char*)buf) + ret));
      }
      return ret;
}
```

# Implementing Pipes



```
/* write to pipe */
ssize_t pipe_write(void* fd_struct, void* buf, size_t count) {
        struct pipe_struct* p = (struct pipe_struct*)fd_struct;
        size_t ret = 0;
        while(ret < count &&
                    (!cb_is_full(p) || !(p->flags[1] & O_NONBLOCK)) {
                while(cb_is_full(p)) {
                    /* blocking write, wait for data to leave */
                    interruptible_wait();
                }
                cb_put(p, *(((char*)buf) + ret));
        }
        return ret;
}
```

# Dispatching Pipes

```
/* service call handling reads */
ssize_t sys_read(int fildes, void* buf, size_t count) {
        struct file_descriptor* fd;
        if(!(fd = get_file_descriptor(fildes))) {
                /* not a valid file descriptor */
                return -EBADF;
        }
        if(fd->read_fn) {
                /* valid read file descriptor, dispatch function */
                return fd->read_fn(fd->fd_struct_data, buf, count);
        }
        return -EBADF;
}
```

# Dispatching Pipes



```c
/* service call handling writes */
ssize_t sys_write(int fildes, const void* buf, size_t count) {
	struct file_descriptor* fd;
	if(!(fd = get_file_descriptor(fildes))) {
		/* not a valid file descriptor */
		return -EBADF;
	}
	if(fd->write_fn) {
		/* valid write file descriptor, dispatch function */
		return fd->write_fn(fd->fd_struct_data, buf, count);
	}
	return -EBADF;
}
```